Методичка 7.1 - Введение в антиотладочные приемы

Общая информация

Антиотладочный прием – это способ обнаружения того, что программа выполняется под управлением отладчика. Основной целью является замедление или предотвращение процесса реверс - инжиниринга.

Также можно сказать, что антиотладочные приемы, это техники, которые противодействуют отладке. Программа понимает, что ее пытаются отладить, изучить и она детектирует это и как следствие, как-то реагирует, например, завершает свою работу.

Антиотладочные приемы очень часто используются во вредоносном ПО. А как известно, с различными видами малвари очень много работают в антивирусных лабораториях. И если вы захотите стать вирусным аналитиком и изучать различное вредоносное ПО, то вам необходимо хорошо разбираться в том, как работают антиотладочные приемы для того, чтобы научиться их обходить.

Есть примеры легального использования антиотладочных приемов, например, их могут использовать разработчики проприетарного ПО.

Основная цель - это усложнить работу злоумышленнику, который будет пытаться восстановить алгоритм проверки ключа для создания кейгена.

Злоумышленнику придется потратить гораздо больше времени и сил, чтобы восстановить алгоритм проверки ключа, если будут использованы различные техники анти - отладки. Возможно дешевле будет ее даже купить, чем тратить время на взлом.

Защита от статического анализа

Антиотладочные приемы могут быть направлены как на борьбу со статическим анализом, так и с динамическим.

Когда мы говорим о статическом анализе, то подразумеваем анализ программы без ее запуска, мы просто анализируем файл, а при динамическом анализе подразумевается запуск программы под отладчиком и изучение ее в процессе работы.

Наиболее распространенные способы защиты от статического анализа:

- обфуская кода
- использование упаковщиков (они же пакеры)

Обфускация кода может быть выполнена перед компиляцией и во время. Упаковщики используются для уже скомпилированных программ.

Если говорить об обфускации кода перед компиляцией, то это прежде всего замена имен переменных и функций на случайные.

Обфуская кода перед компиляцией

Было:

```
...
int packetLength = 255;
char *getPacket(char *);
...
```

Стало:

```
...
int asdkajr1dfa = 255;
char *asdfljakbw23fad(char *);
...
```

Вы можете видеть участок кода до обфускации и после. Код успешно скомпилируется и программа будет работать, но при анализе названия переменных и функций уже будут бесполезны. Как вы помните, названия переменных и функций могут быть в составе бинарного файла, если программа была скомпилирована с отладочными символами. Если отладочные символы не включены в программу, то такая обфускация бесполезна.

Обфуская кода в процессе компиляции

Оптимизация кода средствами компилятора:
Оптимизация по размеру
gcc: -Os
cl: /O1
Оптимизация по скорости
gcc: -O3
cl: /O2

Можно сказать, что применение оптимизации тоже является антиотладочным приемом, так как после оптимизации код гораздо сложнее читать. Компилятор старается заменить инструкции на наиболее оптимальные и количество этих инструкций может сильно вырасти и запутать того, кто будет выполнять анализ дизассемблированного кода программы. Вполне можно допустить, что несколько десятков инструкций в коде программы могут быть заменены несколькими сотнями инструкций просто из-за того, что они будут быстрее выполняться.

Применение упаковщиков

Самая распространенная и достаточно эффективная защита от статического анализа - это использование упаковщиков, например, UPX.

В рамках этого урока у нас еще будет видео, в котором, мы посмотрим, как работает UPX на практике, а сейчас важно понять, принцип работы всех упаковщиков.

Как вы понимаете, если мы просто поиздеваемся над кодом и модифицируем его до неузнаваемости, то такой код скорее всего работать не будет, а если и будет, то некорректно. Упаковщики являются универсальными инструментами, поэтому при упаковке они не будут разбираться как устроена ваша программа. Принцип работы у них очень простой, они берут код вашей программы, без PE-заголовка и прочего мусора, только код и данные и выполняют их шифрование. Алгоритм шифрования может быть абсолютно любой, от примитивной операции XOR с каждым байтом до использования симметричных алгоритмов, например, AES. Итак, пакер сгенерировал какой-то ключ и на этом ключе зашифровал код программы. В таком виде программа, конечно, работать не будет. Для того, чтобы она смогла работать, к зашифрованному блоку добавляют код распаковщика и PE-заголовок, в котором точка входа будет указывать не на наш зашифрованный код, а на распаковщик. Ключ, кстати, на котором был зашифрован наш код, также будет содержаться в распаковщике.

Если мы откроем упакованный бинарный файл, то из кода там будет только код распаковщика и всё. Поэтому статический анализ запакованного файла особого смысла не имеет.

Когда мы запускаем запакованную программу, то сначала начинает работать упаковщик, который выполняет расшифровку кода на ключе, который у него есть, а данные, которые он расшифровал, он размещает где-то в памяти и затем передает на них управление и код программы успешно выполняется.

Защита от динамического анализа

Теперь давайте рассмотрим техники, которые используются для защиты от динамического анализа. Техник, которые препятствуют динамическому анализу очень много, но сводятся они все примерно к следующему:

- обнаружение отладчика
- обнаружение виртуальной машины

Согласитесь, что обычные пользователи, скорее всего, не будут запускать какой-нибудь платный видео-плеер на виртуальной машине. Они, скорее всего, даже не будут знать, что такое виртуальная машина. Поэтому запуск программы на виртуальной машине тоже является косвенным признаком, что вашу программу исследуют. Могут изучать различные взаимодействия с файлами, с сетью, например, с реестром Windows. На виртуальной машине это делать гораздо удобнее.

Ну и, конечно, самой главной целью является отладчик. Практически все анти-отладочные приемы направлены на обнаружение отладчика.

В следующих видео мы подробно разберем несколько способов анти-отладки, а наиболее интересные техники попробуем реализовать в тестовых программах. Ну и, конечно, потом не менее подробно разберем как их можно обойти.